Proving Program Termination with Matrix Weighted Digraphs

Aaron Dutle

NASA Langley Research Center

28th Cumberland Conference on
Combinatorics, Graph Theory & Computing
May 15, 2015

Where I work

LANGLEY FORMAL METHODS

- "Formal Methods" refers to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems.
- Formal methods provide a means to symbolically examine the entire state space of a digital design (hardware or software) and establish correctness or safety properties that are true for all possible inputs.

What I do

Welcome to the PVS Specification
and Verification System

- PVS is a tightly coupled **specification language** and **interactive theorem-prover** used extensively by the formal methods group.

Termination in PVS

Prove termination in two steps.

- Provide a function on the inputs into a well-founded order. (A WFO is a set S and a relation $<$ with no infinite decreasing chain.)
- Show that every recursive call "lowers" the value of the function.

An Example

For $m, n \in \mathbb{N}$, let

$$Ack(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ Ack(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ Ack(m-1, Ack(m, n-1)) & \text{otherwise.} \end{cases}$$

Three calls, so need some measure where:

- $(m,n) > (m-1,1)$,
- $(m,n) > (m-1, Ack(m,n-1))$,
- $(m,n) > (m,n-1)$.

An Example

For $m, n \in \mathbb{N}$, let

$$Ack(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ Ack(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ Ack(m-1, Ack(m, n-1)) & \text{otherwise.} \end{cases}$$

Three calls, so need some measure where:

- $(m,n) > (m-1,1)$,
- $(m,n) > (m-1, Ack(m,n-1))$,
- $(m,n) > (m,n-1)$.

An Example

For $m, n \in \mathbb{N}$, let

$$Ack(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ Ack(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ Ack(m-1, Ack(m, n-1)) & \text{otherwise.} \end{cases}$$

Three calls, so need some measure where:

- $(m,n) > (m-1,1)$,
- $(m,n) > (m-1, Ack(m, n-1))$,
- $(m,n) > (m, n-1)$.

An Example

For $m, n \in \mathbb{N}$, let

$$Ack(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ Ack(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ Ack(m-1, Ack(m,n-1)) & \text{otherwise.} \end{cases}$$

Three calls, so need some measure where:

- $(m,n) > (m-1,1)$,
- $(m,n) > (m-1, Ack(m,n-1))$,
- $(m,n) > (m,n-1)$.

An Example

For $m, n \in \mathbb{N}$, let

$$Ack(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ Ack(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ Ack(m-1, Ack(m, n-1)) & \text{otherwise.} \end{cases}$$

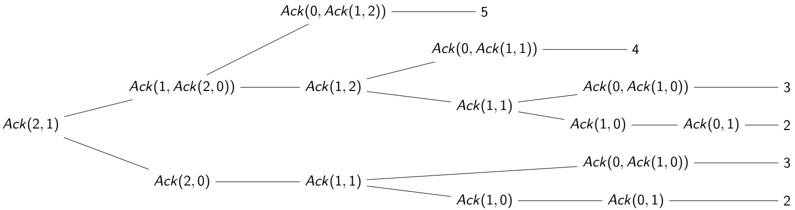
Three calls, so need some measure where:

- $(m,n) > (m-1,1)$,
- $(m,n) > (m-1, Ack(m,n-1))$,
- $(m,n) > (m,n-1)$.

Lexicographic order on pairs works...

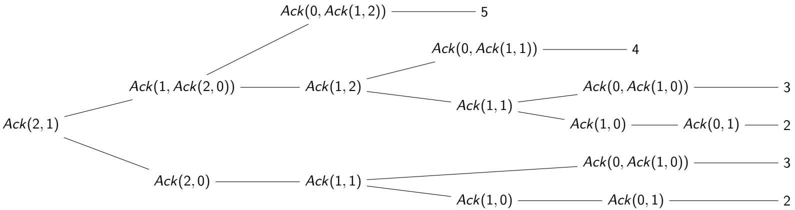
The Size Change Principle

"A program teminates on all inputs if any infinite call sequence would give rise to an infinite descent in some (well-founded) data values." [Lee, Jones, Ben-Amram]



The Size Change Principle

"A program teminates on all inputs if any infinite call sequence would give rise to an infinite descent in some (well-founded) data values." [Lee, Jones, Ben-Amram]

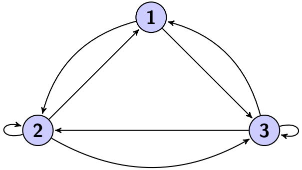


Calling Context Graph for Ackermann

$$Ack(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ Ack(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ Ack(m-1, Ack(m, n-1)) & \text{otherwise.} \end{cases}$$

Three calling contexts:

1. $\{(m,n),(m>0 \wedge n=0),(m-1,1)\}$
2. $\{(m,n),(m>0 \wedge n>0),(m-1,Ack(m,n-1))\}$
3. $\{(m,n),(m>0 \wedge n>0),(m,n-1)\}$



Calling Context Graphs

(Very informally,)
"If every infinite walk on the CCG of a function results in the infinite descent of some well-founded measure, then the function terminates on all inputs." [Manolios and Vroon]

Matrix Weighted Digraphs [Avelar, Muñoz, Rincón]

A framework built on CCGs to efficiently handle several measures.

- Each edge from a CCG is assigned an $N \times N$ matrix with entries in $\{-1, 0, 1\}$.
- Matrix multiplication is standard, but with a non-standard operations on elements.
- The *weight* of a walk on the graph is the product of the matrices on the edges.
- A matrix is called *positive* if it has a 1 entry on the main diagonal.

Matrix Weighted Digraphs [Avelar, Muñoz, Rincón]

A framework built on CCGs to efficiently handle several measures.

- Each edge from a CCG is assigned an $N \times N$ matrix with entries in $\{-1, 0, 1\}$.
- Matrix multiplication is standard, but with a non-standard operations on elements.
- The *weight* of a walk on the graph is the product of the matrices on the edges.
- A matrix is called *positive* if it has a 1 entry on the main diagonal.

A Theorem and a Problem

Theorem (Avelar, Muñoz, Rincón)

If every circuit of a Matrix-Weighted Digraph has positive weight, then the corresponding program terminates on all inputs.

Problem: There are infinitely many circuits, and circuits can be arbitrarily long. How can this be checked?

A Theorem and a Problem

Theorem (Avelar, Muñoz, Rincón)

If every circuit of a Matrix-Weighted Digraph has positive weight, then the corresponding program terminates on all inputs.

Problem: There are infinitely many circuits, and circuits can be arbitrarily long. How can this be checked?

A Theorem and a Problem

Theorem (Avelar, Muñoz, Rincón)

If every circuit of a Matrix-Weighted Digraph has positive weight, then the corresponding program terminates on all inputs.

Problem: There are infinitely many circuits, and circuits can be arbitrarily long. How can this be checked?

One Solution

Theorem

It suffices to examine a finite collection of circuits.

Specifically, if G is the matrix weighted digraph, and the matrices are $N \times N$, checking circuits with length at most $3^{N^2}|G| + 1$ suffices.

Proof.

One Solution

Theorem

It suffices to examine a finite collection of circuits.

Specifically, if G is the matrix weighted digraph, and the matrices are $N \times N$, checking circuits with length at most $3^{N^2}|G| + 1$ suffices.

Proof.

$\square$

One Solution

Theorem

It suffices to examine a finite collection of circuits.

Specifically, if G is the matrix weighted digraph, and the matrices are $N \times N$, checking circuits with length at most $3^{N^2}|G| + 1$ suffices.

Proof.

$\square$

A Process

Idea:

- Let $S_i = \{L_v | v \in G\}$, where L_v contains all matrices that are the weight of some circuit at v with length at most i.
- Start with empty lists for S_0.
- Calculate S_{i+1} from S_i.

A Process

Idea:

- Let $S_i = \{L_v | v \in G\}$, where L_v contains all matrices that are the weight of some circuit at v with length at most i.
- Start with empty lists for S_0.
- Calculate S_{i+1} from S_i.

A Process

Idea:

- Let $S_i = \{L_v | v \in G\}$, where L_v contains all matrices that are the weight of some circuit at v with length at most i.
- Start with empty lists for S_0.
- Calculate S_{i+1} from S_i.

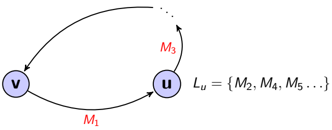
A Process

Idea:

- Let $S_i = \{L_v | v \in G\}$, where L_v contains all matrices that are the weight of some circuit at v with length at most i.
- Start with empty lists for S_0.
- Calculate S_{i+1} from S_i. $\leftarrow$ The hard part.

The Hard Part

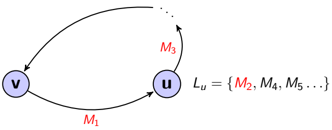
Given a *cycle* at v, instead of multiplying matrices only from the edges, for each vertex u on the cycle, include a matrix from L_u.
Simulates following a circuit at u.



The Hard Part

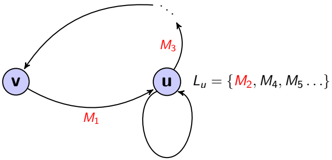
Given a *cycle* at v, instead of multiplying matrices only from the edges, for each vertex u on the cycle, include a matrix from L_u.

Simulates following a circuit at u.



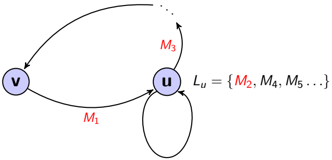
The Hard Part

Given a *cycle* at v, instead of multiplying matrices only from the edges, for each vertex u on the cycle, include a matrix from L_u. Simulates following a circuit at u.



The Hard Part

Given a *cycle* at v, instead of multiplying matrices only from the edges, for each vertex u on the cycle, include a matrix from L_u. Simulates following a circuit at u.



Append the result to L_v. Do this for every vertex, cycle at the vertex, and choice of matrices at vertices of the cycle.

An Optimization

The lists L_v can get long, making the calculation of S_{i+1} slow. We can do better.

- Matrices form a partial order under pointwise $\leq$.
- Multiplication respects the partial order.

Instead of keeping *all* matrices in L_v, keep only those *minimal* with respect to this partial order.

An Optimization

The lists L_v can get long, making the calculation of S_{i+1} slow. We can do better.

- Matrices form a partial order under pointwise $\leq$.
- Multiplication respects the partial order.

Instead of keeping *all* matrices in L_v, keep only those *minimal* with respect to this partial order.

An Optimization

The lists L_v can get long, making the calculation of S_{i+1} slow. We can do better.

- Matrices form a partial order under pointwise $\leq$.
- Multiplication respects the partial order.

Instead of keeping *all* matrices in L_v, keep only those *minimal* with respect to this partial order.

Early Exits

A few properties of the (optimized) process.

- If the process ever results in a non-positive matrix, it can quit. (Failed to prove termination...)
- If ever $S_{i+1} = S_i$, then every further iteration will equal S_i. (Stabilization...)
- The process will *always* stabilize. (At worst $3^{N^2}|G| + 1$ iterations.)

Early Exits

A few properties of the (optimized) process.

- If the process ever results in a non-positive matrix, it can quit. (Failed to prove termination...)
- If ever $S_{i+1} = S_i$, then every further iteration will equal S_i. (Stabilization...)
- The process will *always* stabilize. (At worst $3^{N^2}|G| + 1$ iterations.)

Early Exits

A few properties of the (optimized) process.

- If the process ever results in a non-positive matrix, it can quit. (Failed to prove termination...)
- If ever $S_{i+1} = S_i$, then every further iteration will equal S_i. (Stabilization...)
- The process will *always* stabilize. (At worst $3^{N^2}|G| + 1$ iterations.)

Early Exits

A few properties of the (optimized) process.

- If the process ever results in a non-positive matrix, it can quit. (Failed to prove termination...)
- If ever $S_{i+1} = S_i$, then every further iteration will equal S_i. (Stabilization...)
- The process will *always* stabilize. (At worst $3^{N^2}|G| + 1$ iterations.)

Terminal Remarks

In practice, the process always stabilizes early.

Example

For $Ack(m, n)$, let $\mu_1(m, n) = m$ and $\mu_2(m, n) = n$.

Terminal Remarks

In practice, the process always stabilizes early.

Example

For $Ack(m, n)$, let $\mu_1(m, n) = m$ and $\mu_2(m, n) = n$.

The guarantee is $3^5 + 1 = 244$ iterations.

Terminal Remarks

In practice, the process always stabilizes early.

Example

For $Ack(m, n)$, let $\mu_1(m, n) = m$ and $\mu_2(m, n) = n$.

The guarantee is $3^5 + 1 = 244$ iterations.

The process stabilizes after 2 iterations.

Thanks!